P1235R0: Implicit constexpr

ISO/IEC JTC1 SC22/WG21 - Programming Languages - C++

Authors:
Bryce Adelstein Lelbach <brycelelbach@gmail.com>
Hana Dusíková <hana.dusikova@avast.com>

Audience:
Evolution Working Group (EWG)

Motivation

Over the course of the last 8 years, constexpr has evolved and proliferated throughout the C++ standard library and wider C++ ecosystem. Over time, constexpr restrictions have been relaxed as we’ve realized that the original restrictions were too conservative, compiler technology has matured, and the benefits of constexpr for compile time programming became apparent.

As we continue to expand the subset of C++ that is allowed in constexpr code:
- The quantity of functions that cannot be constexpr is decreasing.
- The quantity of functions we want to use in constant expressions is increasing.

In C++17, we took a step towards making constexpr the default when we started implicitly treating lambda call operators as constexpr. While this is an improvement, there is now an artificial inconsistency between functions and lambdas.

Consider the following code:

```cpp
auto add0 = [] (int a, int b) { return a + b; };
auto add1(int a, int b) { return a + b; }

constexpr int x = add0(17, 42);
constexpr int y = add1(17, 42); // COMPIL FAILURE.
```

The need to manually annotate functions as constexpr is starting to become burdensome, both within the C++ standard library and in 3rd party C++ libraries.
**Design**

We propose that when a function is called in a constant expression, if it is not marked as `constexpr`, and it is defined in the current translation unit, it should be treated as if it was declared `constexpr`.

```cpp
double reciprocal(int v) {
  if (v == 0) throw invalid_argument("divide by zero");
  else return 1.0 / v;
}
```

```cpp
constexpr double w = reciprocal(0); // COMPILe FAILURE.
constexpr double x = reciprocal(2); // Ok.
double y = reciprocal(0); // Throws at runtime.
double z = reciprocal(2); // Ok.
```

However, an opt-out mechanism is needed to ensure that library designers can prevent users from relying on their functions being implicitly `constexpr`. For example, suppose I had this function in my library:

```cpp
auto add(array<int, 4> a, array<int, 4> b) {
  for (int i = 0; i < 4; ++i)
    a[i] += b[i];
  return a;
}
```

```cpp
constexpr array<int, 4> a = ...;
constexpr array<int, 4> b = ...;
```

```cpp
array<int, 4> c = add(a, b);
// Not implicitly constexpr.
```

```cpp
constexpr array<int, 4> c = add(a, b);
// Implicitly treated as constexpr, ok.
```

Under the proposed implicit `constexpr` mechanism, this function could be called in constant expressions. If users of this function started to take advantage of this, I would be unable to later change this function in a way that made it impossible to evaluate as `constexpr`:
auto add(array<int, 4> a, array<int, 4> b)
{
    // __simd_add is a non-constexpr extern function.
    __simd_add(a.data(), b.data());
    return a;
}

constexpr array<int, 4> a = ...;
constexpr array<int, 4> b = ...;

array<int, 4> c = add(a, b);
// Not implicitly constexpr.

constexpr array<int, 4> c = add(a, b);
// Implicitly treated as constexpr, COMPILe FAILURE.

To prevent a function from being implicitly treated as constexpr, we propose allowing a function author to opt-out with a syntax such as:

constexpr(false) auto add(array<int, 4> a, array<int, 4> b);

This syntax could also be used to express a desire for a function to be callable only from constant expressions - constexpr(true) - similar to the proposed constexpr!

A summary of how constexpr specifiers would work with the proposed changes:

<table>
<thead>
<tr>
<th>No specifier</th>
<th>Can be called in a constant expression as if it was declared as a constexpr function and has a definition in this translation unit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>constexpr</td>
<td>Works as it does today.</td>
</tr>
<tr>
<td>constexpr(false)</td>
<td>Cannot be called in a constant expression.</td>
</tr>
<tr>
<td>constexpr(true)</td>
<td>Can only be called in constant expressions.</td>
</tr>
</tbody>
</table>